Upgrading HTTPS in mid-air:
An empirical study of strict transport security and key pinning
WORKING DRAFT, currently under peer review

Michael Kranch
Princeton University
mkranch@princeton.edu

Joseph Bonneau
Princeton University
jbonneau@princeton.edu

***Abstract*—We have conducted the first in-depth empirical study of two important new web security features, strict transport security (HSTS) and public-key pinning. Both have been added to the web platform to harden HTTPS, the prevailing standard for secure web browsing. While HSTS is further along, both features still have very limited deployment at a few large websites and a long tail of small security-conscious sites. We find evidence of developers not understanding the correct use of these features, with a substantial portion using them in invalid or illogical ways. We also identify a number of subtle but important errors in practical deployments which often undermine the security these new features are meant to provide. For example, the majority of pinned domains undermine the security benefits by loading non-pinned resources with the ability to hijack the page. A substantial portion of HSTS domains and nearly all pinned domains leaked cookie values, including login cookies, due to the poorly-understood interaction between HTTP cookies and the same-origin policy. Our findings highlight that the web platform, as well as modern web sites, are large and complicated enough to make even conceptually simple security upgrades challenging to deploy in practice.**

I. Introduction

HTTPS [1], which consists of layering HTTP traffic over the TLS/SSL encrypted transport protocols [2] to ensure confidentiality and integrity, is the dominant protocol used to secure web traffic. Though there have been many subtle cryptographic flaws in TLS itself (see [3] for an extensive survey), the most significant problem has been inconsistent and incomplete deployment of HTTPS. Browsers must seamlessly support a mix of HTTP and HTTPS connections, enabling *stripping attacks* [4]) whereby network attackers attempt to downgrade a victim's connection to insecure HTTP despite support for HTTPS at both the server and client.

The primary countermeasure to HTTPS stripping is *strict transport security* (HSTS) [5] through which browsers learn that specific domains must only be accessed via HTTPS. This policy may be specified dynamically by sites using an HTTP header, or preloaded by browsers for popular domains as Google Chrome and Mozilla Firefox now do.

While HSTS is conceptually very simple, deployment is still sparse and there are subtle interactions with other browser features. In particular, the interaction between domains and subdomains can be complicated as can the interaction between the same-origin policy and protections for HTTP cookies. This can lead to a number of deployment errors which enable an attacker to steal sensitive data without compromising HTTPS itself.

Beyond HTTPS stripping, there are growing concerns about weaknesses in the certificate authority (CA) system. The public discovery of commercial software to use *compelled certificates* to perform network eavesdropping attacks [6], as well as high-profile compromises of several trusted CAs, have spurred interest in defending against a new threat model in which the attacker may obtain a *rogue certificate* for a target domain signed by a trusted CA.

While many protocols have been proposed to maintain security against an attacker with a rogue certificate for a target domain signed by a trusted CA [3], the only defense deployed to date is *public-key pinning* (or just *key pinning*), by which a browser learns to only connect to specific domains over HTTPS if one of a designated set of keys is used. This policy can be used to "pin" a domain to a whitelist keys of which at least one must appear somewhere in the server's certificate chain. This can be used to pin a domain to potential end-entity keys, certificate authorities, or a mix of both. Key pinning is currently deployed only as a browser-preloaded policy with Chrome, though support from Firefox is imminent as is support for a header-based mechanism for domains to dynamically declare pins [7].

While both technologies are still in the early stages, there is now sufficient deployment that we can draw some meaningful conclusions from their use in practice. In this work we provide the first study the deployment of both HSTS and key pinning. Our two main contributions are:

- A comprehensive, measurement-based survey of HSTS and key pinning. We study both the list of domains with preloaded security policies in Firefox and Chrome and the 10,000 most highly-visited domains as provided by Alexa [8]. We employ a modified Selenium web browser for realistic crawling (see Section III) and are able to examine both a web page's static code as well as dynamically observe traffic initiated by the website as it is loaded. Our code will ultimately be released as an open-source tool for security scanning.
- A catalog of security bugs observed in practice (Sections IV–VII). A summary is provided in Table I. To the best of our knowledge we provide the first published evidence of these bugs' appearance in the wild. Our work is useful for any administrator seeking to deploy HSTS and/or pinning securely.

In summarizing our findings (Section IX) we highlight

several underlying causes of these errors. In particular we believe the specification of HSTS and pinning suffers from both insufficient flexibility and a lack of sensible defaults. These lessons are timely given the considerable amount of ongoing research and development of new proposals for upgrading HTTPS security.

II. Overview of web security technologies

The core protocols of the World Wide Web, namely HTTP and HTML, were not designed with security in mind. As a result, a series of new technologies have been gradually tacked on to the basic web platform. All of these are to some extent weakened by backwards-compatibility considerations. In this section we provide an overview of relevant web security concepts which we will study in this paper.

A. HTTPS and TLS/SSL

HTTPS is the common name for "HTTP over TLS" [1] which combines normal HTTP traffic with the TLS [2], [9], [10] (Transport Layer Security) protocol instead of basic (insecure) TCP/IP. Most HTTPS implementations will also use the older SSL v3.0 (Secure Sockets Layer) protocol [11] for backwards compatibility reasons although it contains a number of cryptographic weaknesses. TLS and SSL are often used interchangeably to describe secure transport; in this paper we'll strictly refer to TLS with the understanding that our analysis applies equally to SSL v3.0.

The goals of TLS are confidentiality against eavesdroppers, integrity against manipulation by an active network adversary, and authenticity by identifying one or both parties with a certificate. The main adversary in TLS is typically called a *man in the middle* (MitM) or *active network attacker*, a malicious proxy who can intercept, modify, block, or redirect all traffic. Formally, this is referred to as a Dolev-Yao attacker model [12]. We will not discuss cryptographic issues with TLS; Clark and van Oorschot provide a thorough survey [3].

1) Certificates and Certification Authorities: The ultimate goal of HTTPS is to bind the communication channel to the legitimate server for a given web domain. This is achieved with the use of server certificates in TLS.[1]

Names are bound at the domain level, also sometimes referred to as "hostname," "host" or "fully-qualified domain name". In this paper, we'll always use "domain" to refer to a fully-qualified domain name. We'll use the term "base domain" to refer to the highest-level non public domain in a fully-qualified domain name, also sometimes referred to as "public suffix plus 1" (PS+1).[2] For example, for `www.a.com` the base domain is `a.com`.

HTTPS clients will check that the "common name" field in the server's presented certificate matches the domain for each HTTP request. The exact rules are somewhat complicated [13] enabling the use of wildcards and the "subject alternative name" extension to allow certificates to match an arbitrary number of domains.

If name matching fails, or a certificate is expired, malformed, or signed by an unknown or untrusted certificate authority, HTTPS clients typically show the user a warning. Over time browser vendors have made the warnings more intrusive and click-through rates have declined significantly [14], [15] and are now below 50% for Mozilla Firefox. This is generally considered a positive development as the vast majority of HTTPS warning messages represent false positives due to server misconfigurations or expired certificates [16], [17].

B. Strict transport security

Because a large portion of web sites only support insecure HTTP, user agents must support both HTTP and HTTPS. Many domains serve traffic over both HTTP and HTTPS. This enables an active network attacker to attempt to downgrade security to plain HTTP by intercepting redirects from HTTP to HTTPS or rewriting URLs contained in an HTTP page to change the protocol from HTTPS to HTTP. Such an attack is called an *SSL stripping* or *HTTPS stripping* attack. While the threat has long been known [18], it gained increased attention in 2009 when the popular *sslstrip* software package was released to automate the attack [4].

Browsers use special UI to indicate to the user whether a connection is made over HTTP or HTTPS. Typically, this UI includes showing `https` in the browser's address bar and showing a padlock icon. However, user studies have indicated that the vast majority of users (upwards of 90%) don't notice if these security indicators are missing and are still willing to transmit sensitive data such as passwords or banking details [19]. Thus, it is insufficient to rely on users to detect if their connection to a normally-secure server has been downgraded to HTTP by an attacker.

To counter the threat of HTTPS stripping, in 2008 Jackson and Barth proposed "ForceHTTPS" [20] to enable servers to request clients only communicate over HTTPS. Their proposal was ultimately renamed "HTTP strict transport security" and standardized in RFC 6797 [5].[3]

1) HSTS security model: HSTS works as a binary (per domain) security policy. Once set, the user agent must refuse to send any traffic to the domain over plain HTTP. Any request which would otherwise be transmitted over HTTP (for example, if the user clicks on a link with the `http` scheme specified) will be upgraded from HTTP to HTTPS.

In addition to upgrading all traffic to HTTPS, the HSTS specification recommends two other changes. First, any TLS error (including certificate errors) should result in a *hard fail* with no opportunity for the user to ignore the error. Second, it is recommended (non-normatively) that browsers disable the loading of insecure resources from an HSTS-enabled page; this policy has since been adopted by Chrome and Firefox for all HTTPS pages even in the absence of HSTS (see Section V).

[1] TLS also supports mutual authentication in which both client and server are identified with a long-term certificate. However, in nearly all use on the web only the server is authenticated and the client authentication is left for higher-level protocols (typically passwords submitted over HTTPS).

[2] Detecting base domains is not as simple as finding the domain immediately below the top-level domain (TLD+1) due to the existence of *public suffixes* of more than one domain, such as `.ca.us` or `.ac.uk`. We use Mozilla's public suffix list (https://publicsuffix.org/) as the canonical list.

[3] Jackson and Barth [20] originally proposed servers would request HSTS status by setting a special cookie value, but this was ultimately changed to be an HTTP header.

TABLE I. SUMMARY OF MAIN VULNERABILITIES FOUND

error	Section	prevalence %	prevalence #		security implications
Preloaded HSTS without dynamic HSTS	IV-E	41.3%	161/390	domains with preloaded HSTS	HTTPS stripping possible on browsers besides Chrome
Erroneous HSTS headers set	IV-E	27.7%	44/159	domains attempting to set HSTS	HTTPS stripping possible
Pinned site with non-pinned active content	V	1.9% 50.0%	5/268 4/8	pinned base domains non-Google pinned base domains	page hijacking, data theft with a rogue certificate
Pinned site with non-pinned passive content	V	60.8% 25.0%	163/268 2/8	pinned base domains non-Google pinned base domains	page modifications with a rogue certificate
Cookies scoped to non-pinned subdomains	VI-C	2.2% 50.0%	6/268 4/8	base domains with preloaded pins non-Google pinned base domains	cookie theft with a rogue certificate
Cookies scoped to non-HSTS subdomains	VI-B	23.7%	61/257	base domains with preloaded HSTS	cookie theft by active network attacker
`http` links to external HSTS domain	VII	61.8% 12.9%	181/ 293 2204/17086	base domains with preloaded HSTS links to external HSTS domains	HTTPS stripping of initial connections

By default, HSTS is declared for a specific fully-qualified domain name, though there is an optional `includeSubDomains` directive which applies to all subdomains of the domain setting the policy. For example, if `a.com` sets an HSTS policy with `includeSubDomains`, then all traffic to `a.com` as well as `b.a.com` and `d.c.b.a.com` must be over HTTPS only.[4]

Note that while HSTS requires a valid TLS connection, it places no restrictions on the set of acceptable certificates beyond what the user agent would normally enforce. That is, HSTS simply requires *any valid TLS connection* with a trusted certificate. It is not designed as a defense against insecure HTTPS due to rogue certificates (see Section II-C), only against the absence of HTTPS completely.

2) HSTS headers: The primary means for a server to establish HSTS is by setting the HTTP header [21] `Strict-Transport-Security`. Compliant user agents will apply an HSTS policy to a domain once the header has been observed over an HTTPS connection with no errors. Note that setting the header over plain HTTP has no effect; although a number of sites do so anyways (see Section IV-E).

In addition to the optional `includeSubDomains` directive, an HSTS header must specify a `max-age` directive instructing the user agent on how long to cache the HSTS policy. This value is specified in seconds and represents a commitment by the site to support HTTPS for at least that time into future. It is possible to "break" this commitment by serving an HSTS header with `max-age=0`, this must itself be served over HTTPS.

For regularly-visited HSTS domains (at least once per `max-age` period), the policy will be continually updated and prevent HTTP traffic indefinitely. This can be described as a *continuity* policy.[5] A known shortcoming of HSTS is that it can't protect initial connections or connections after extended inactivity or flushing of the browser's HSTS state. HSTS policy caching may also be restricted by browser privacy concerns, for example policies learned during "private browsing" sessions should be discarded because they contain a record of visited domains.

3) HSTS preloads: To address the vulnerability of HTTPS stripping before the user agent has visited a domain and observed an HSTS header, Chrome now ships with a hard-coded list of domains receiving a pre-loaded HSTS policy[6] and Firefox is planning to ship a similar list of preloaded domains shortly. This approach reduces security for pre-loaded domains to maintaining an authentic, up-to-date browser installation.

Preloaded domains receive an automatic HSTS policy from the browser and may optionally specify `includeSubDomains`.[7] There is no per-domain `max-age` specification; however in Chrome's implementation the entire preload list has an expiration date if the browser is not regularly updated.

4) HTTPS Everywhere: The EFF's HTTPS Everywhere browser extension[8] (available for Chrome and Firefox) provides similar protection for a much larger list (currently over 5,000 domains). It has been available since 2011. HTTPS Everywhere extension relies on a large group of volunteers to curate the preloaded list in a distributed manner. Because it is an optional (though popular) browser extension, HTTPS Everywhere is willing to tolerate occasional over-blocking errors in return for increased coverage compared to the more conservative Chrome preload list. Because HTTPS everywhere is crowd-sourced, errors are due to the developers and not site operators themselves; hence we won't study it in this work.

C. Key pinning

HSTS is useful for forcing traffic to utilize HTTPS; however, it has no effect against an attacker able to fraudulently obtain a signed certificate for a victim's domain (often called a *rogue certificate*) and use this in a man-in-the-middle attack. Because every trusted root in the browser can sign for any domain, an attacker will succeed if they are able to obtain a rogue certificate signed by *any* trusted root (of which there are hundreds [23]–[25]). This vulnerability has long been known and security researchers have been obtained several rogue certificates by exploiting social engineering and other flaws in the certificate authority's process for validating domain ownership [3].

However, in 2010 it was reported publicly for the first time that commercial software was available for sale to government

[4]Note that `includeSubDomains` covers subdomains to arbitrary depth. This is unlike the semantics of wildcard certificates, which only apply to one level of subdomain [13].

[5]HSTS can also be described as a "trust-on-first-use" (TOFU) scheme. We prefer the more general term continuity which does not imply permanent trust after first use.

[6]A small number of domains are designated to only receive HSTS protection if the client TLS library supports the Server Name Indication (SNI) extension [22]. Because most web clients now support SNI, in this paper we evaluate these preloaded policies similarly to the others.

[7]The syntax between HSTS preloads and HSTS headers is unfortunately incompatible, with `includeSubDomains` specified in the former and `include_subdomains` used in the latter, among other minor details we will omit for clarity.

[8]https://www.eff.org/https-everywhere

agencies to utilize rogue certificates to intercept traffic en masse [6]. This raised the concern of governments using *compelled certificates* obtained by legal procedures or extralegal pressure to perform network eavesdropping attacks.

In addition to the risk of government pressure, a number of high-profile CA compromises have been detected since 2011[9] [26] including security breaches at Comodo and DigiNotar (which has since been removed as a trusted CA from all browsers) and improperly issued subordinate root certificates from TrustWave and TurkTrust. Collectively, these issues have demonstrated that simply requiring HTTPS via HSTS is not sufficient and steps need to be taken to limit the risk of rogue certificates.

1) Pinning security model: Key pinning specifies a limited set of public keys which a domain can use in establishing a TLS connection. Specifically, a key pinning policy will specify a list of hashes (typically SHA-1 or SHA-256) each covering the complete Subject Public Key Info field of an X.509 certificate. To satisfy a pinning policy, a TLS connection must use a certificate chain where at least one key appearing in the chain matches at least one entry in the pin set. This enables site operators to pin their server's end-entity public key, the key of the server's preferred root CA, or the key of any intermediate CA. Pinning makes obtaining rogue certificates much more difficult, as the rogue certificate must also match the pinning policy which should greatly reduce the number of CAs which are able to issue a usable rogue certificate.

In fact, the browsers' default policy can be viewed as "pinning" all domains with the set of all trusted root certificate authorities. Explicit pinning policies further reduce this set for specific domains. Much like HSTS, pinning policies apply at the domain level but an optional `includeSubDomains` directive extends this protection to all subdomains.

The risk of misconfigured pinning policies is far greater than accidentally setting HSTS. HSTS can be undone as long as the site operator can present any acceptable certificate, whereas if a site declares a pinning policy and then can't obtain a usable certificate chain satisfying the pins (for example, if it loses the associated private key to a pinned end-entity key), then the domain will effectively be "bricked" until the policy expires. For this reason, pinning policies often require a site to specify at least two pins to mitigate this risk.

2) Pinning preloads: Chrome has deployed preloaded pinning policies since 2011, although only a handful of non-Google domains currently participate. Firefox is set to ship support for pinning soon, adding pins for Mozilla properties to Google's basic list. Like with preloaded HSTS, preloaded pinning policies have no individual expiration date but the entire set expires if the browser is not frequently updated. No other browsers have publicly announced plans to implement preloaded pinning.

3) Pinning headers (HPKP): A draft RFC specifies HTTP Public Key Pinning (HPKP) by which sites may declare pinning policies via the `Public-Key-Pins` HTTP header [7]. The syntax is very similar to HSTS, with an optional `includeSubDomains` directive and a mandatory `max-age` directive. Pinning policies will only be accepted when declared via a valid TLS connection which itself satisfies the declared policy.

Unlike HSTS, the HPKP standard adds an additional `Public-Key-Pins-Report-Only` header. When this policy is declared, instead of failing if pins aren't satisfied the user agent will send a report to a designated address. This is designed as a step towards adoption for domains unsure if they will cause clients to lose access. Additionally, the standard recommends that user agents limit policies to 60 days of duration from the time they are declared, even if a longer `max-age` is specified, as a hedge against attackers attempting to brick a domain by declaring a pinning policy which the genuine owner can't satisfy.

Currently, no browsers support HPKP and the standard remains a draft. Chrome and Firefox have both announced plans to support the standard.

III. Measurement setup

To study how HSTS and pinning are deployed in practice, we made use of an automated web measurement platform. Our goal was to measure web sites by simulating real browsing as accurately as possible while also collecting as much data as possible. To this end we used a version of the Firefox browser modified only for automation and data capture. Our approach combined *static analysis* of downloaded page content with *dynamic analysis* of all connections actually made by the browser on behalf of rendered page content.

OpenWPM: We utilized OpenWPM as the backbone for our testing. OpenWPM is an open source web-crawling and measurement utility designed to provide a high level of reproducibility [27]. OpenWPM is itself built on top of the well-known Selenium Automated Browser [28] which abstracts away details such as error handling and recovery from browser crashes.

Static Resources: Selenium provides an interface for inspecting the parsed DOM *after* pages have completed loading. We utilized this interface to extract all tags of interest to check for potential mixed content errors that were not triggered by our browsing. We also examined other elements like links (`a` tags) which have security implications (see Section VII) but do not generate traffic without user action.

Dynamic Resources: By itself, neither Selenium nor OpenWPM contain an API to instrument the browser as it executes scripts and loads resources on behalf of the page. We utilized the Firefox add-on system to build a custom Firefox extension to instrument and record all resource calls as the page was loaded to capture dynamic resource loading. Our extension implements the `nsIContentPolicy` interface in the Firefox extension API, which is called prior to any resource being loaded. This interface is designed to allow extensions to enforce arbitrary content-loading restrictions. In our case, we allowed all resource loads but recorded the target URL, the page origin URL, as well as the context in which that request was made, for example if it was the result of an image being loaded or an XMLHttpRequest (Ajax).

[9]It should be noted that most of these issues were detected due to Chrome's deployment of key pinning. It is possible a large number of CA compromises occurred before pinning was deployed but evaded detection.

Sites accessed: We conducted three main crawls in our study. The first was a depth-one[10] crawl of every domain listed in Chrome and Firefox's preloaded HSTS/pinning lists. We tried fetching both the exact domain and the standard `www` subdomain. Some domains were not accessible, as we discuss in Section IV. We did not attempt to find alternate pages domains without an accessible home page. We did manually identify those which appear to be content-delivery networks without a home-page by design, with the rest being domains that have ceased operation.

The second crawl was an expanded crawl for the limited set of sites with a preloaded pinning policy. For Twitter, Dropbox, and Google domains, we performed this crawl while "logged in," that is, using valid session cookies for a real account. The other pinned domains did not implement a login system. Finally, we performed a simple crawl of all sites in the Alexa top 10,000 [8] sites to test for the presence of an HSTS header.

IV. Current deployment

In this section we provide an overview of current deployment of HSTS and pinning using crawling and inspection of the preloaded lists used by Chrome and Firefox.

A. Preloaded implementations

Chrome's preloaded list is stored in a single JSON file[11] containing both pinning and HSTS policies.[12] The criteria for inclusion in Chrome's list are not formally documented and the process consists of email communication with an individual engineer responsible for maintaining the list [29], [30].

Mozilla's implementation uses two separate files for preloaded HSTS and pinning policies. The preloaded HSTS file is compiled as C code. Currently, the list is a strict subset of Chrome's list based on those domains in Chrome's list which also set an HSTS header with a `max-age` of at least 18 weeks [31]. The `includeSubDomains` parameter must also be set in the header to be preloaded, so 45 domains are preloaded in Chrome with `includeSubDomains` set but preloaded in Firefox without it. Because Mozilla's list is a strict subset of Chrome's list, we perform all testing for the remainder of the paper on the larger Chrome list.

Firefox also has a preloaded pin list which is implemented as a JSON file with an identical schema to Chrome's. Currently, the only entries are 6 Mozilla-operated domains. We perform testing on both Mozilla and Chrome's preloaded pinning lists for the remainder of the paper.

B. Preloaded HSTS

Chrome's preloaded list, launched in 2010, currently consists of 390 domains from 494 unique base domains as summarized in Table II. Roughly half of the currently list represents Google-owned properties.

[10]By depth-one, we mean we visited original page plus followed every link on that page.

[11]Technically, the file is not valid JSON because it includes C-style comments.

[12]Chrome's list also includes one entry for learn.doubleclick.net which sets neither HSTS nor pins and is thus meaningless.

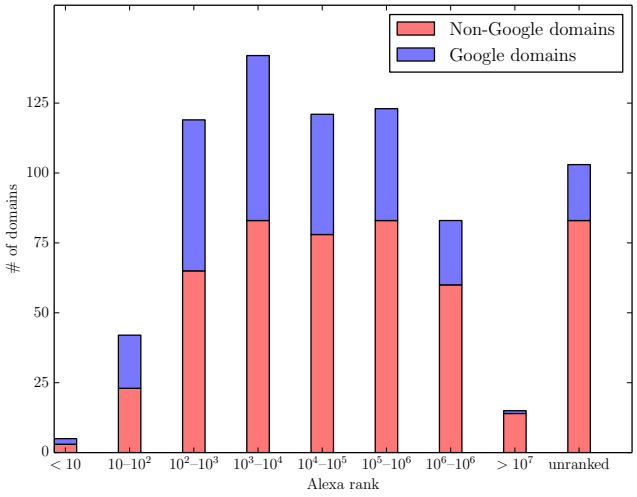

Fig. 1. Histogram of Alexa Site Ranks [8] for domains with a preloaded HSTS policy in Google Chrome.

Beyond the non-Google domains there are a large number of relatively small websites judging by site traffic data from Alexa, as shown in Figure 1. The median site rank is about 100,000 and the average is 1.5M. Additionally, only 5 of the top 100 and 16 of the top 1000 non-Google sites are included in the preloaded HSTS list, suggesting that uptake is primarily driven by sites' security interest and not by their size. At least 12 sites on the list appear to be individual people's homepages.

Even with the relatively small scale and short history of preloaded HSTS, the list is surprisingly stale. Of the 233 non-Google base domains with a preloaded policy, at least 34 domains return a 404 or other site-not-accessible errors. We excluded from this count sites which by manual inspection appeared to be content-delivery networks with no home page by design. These outdated preloaded entries include sites like the infamous liberty.lavabit.com and sunshinepress.org (a long outdated WikiLeaks alias). In addition to the no-longer-available sites, 15 sites redirect directly to a distinct plain-HTTP URL. Thus, of the 233 non-Google base domains in the list, 49 represented policies that are no longer needed, or 21% of the total.

These issues were not limited to non-Google domains either: 5 Google-operated domains also redirect directly to HTTP including ssl.google-analytics.com and talk.google.com. The Google-owned company urchin.com is still in the list although it was discontinued in May 2012.

We also saw 15 preloaded HSTS domains redirecting to non-HSTS domains. For example, https://www.evernote.com is the only Evernote entry in the preloaded list and redirects to https://evernote.com (which does not set a dynamic HSTS header). These may represent stale data, or sites may have a legitimate reason to have a changed branding, but the result is that these preloaded policies have little effect in practice.

While scalability of the preload list is often discussed in terms of the amount of storage required to ship the list and the cost of checking every browser request for a preloaded policy, these findings suggest the far simpler problem of human management of the list may be an issue much sooner.

TABLE II. SUMMARY OF PRELOADED HSTS AND PINNING POLICIES

			Google Chrome						Mozilla Firefox	
			total		*Google*		*non-Google*			
HSTS	Pinned	IncludeSubdomains	total domains	base domains	total domains	base domains	total domains	base domains	total domains	base domains
✓	–	–	128	76	0	0	128	76	140	82
✓	–	✓	194	161	0	0	194	161	143	111
–	✓	–	0	0	0	0	0	0	0	0
–	✓	✓	247	241	239	238	8	3	6	4
✓	✓	–	9	7	5	3	4	4	0	0
✓	✓	✓	59	28	52	24	7	4	0	0
total			638	493	297	261	341	233	289	192

TABLE III. SUMMARY OF PRELOADED PIN SETS

Pin set name	# CA pins	# End-entity pins	total domains	base domains
cryptoCat	1	1	1	1
dropbox	18	0	2	1
google	2	0	295	260
lavabit	0	1	1	1
mozilla	21	0	6	3
mozilla_fxa	1	0	2	1
tor	2	3	5	1
tor2web	1	1	1	1
twitterCDN	42	1	1	1
twitterCom	21	1	6	1

C. Preloaded pinning

A summary of the preloaded pinning policies is shown in Table II. Outside of Google, which pins nearly all of its properties, pinning remains relatively rare with only 25 domains (15 base domains) implementing pinning, including Mozilla's pinning which remains turned on only in experimental builds of Firefox.

Table III providers further details of the domains utilizing pinning and the size of various pin sets being used. Only 3 for-profit websites (Google, Twitter, and Dropbox) are utilizing pinning. Of these, Twitter and Dropbox both pin to a large number of certificate authorities rather than end-entity keys. Mozilla takes a similar approach. This suggests that pinning is still challenging for large websites to deploy as they rely on a huge number of different certificates for different parts of their operation.

By contrast, Google pins to just 2 CAs, although they are intermediate CAs operated by Google itself (Google Internet Authority). Google is rare in controlling a CA as well as largely running its own data centers and content-delivery networks, making it much easier to pin to a smaller set of keys.

The remaining pinned domains are all non-profit entities with much more limited pin sets. Lavabit was the only domain specifying a pinning policy consisting of only an end-entity-key, leaving no certificate authority able to undermine its security. Of course, Lavabit suspended operation in August 2013, indicating that the preloaded pin list is also not completely up-to-date.

Pinning without HSTS: Most pinned domains also set HSTS, which is consistent with the belief that pinning is more complicated to deploy and HSTS defends against easier-to-execute attacks. However, 238 Google domains which are pinned do not set HSTS. While Google has not explained this policy, analysis shows 217 of the domains are country-specific variations of the main Google home-page (such as google.sn) as well as google.com itself. For these pages, which implement Google's search engine, supporting non-HTTPS access has been deemed a policy requirement by Google for schools, libraries, and other locations which require filtering adult content [32]. The remainder are mostly ad network domains (doubleclick.net, googleadservices.com) and content delivery network domains (gstatic.com, googleusercontent.com), but a few security-critical domains are not protected with HSTS including googleapis.com and android.com. This suggests that in some cases enabling pinning may actually be less difficult than HSTS due to the need to support legacy clients which cannot use HTTPS.

Twitter and Tor2web are the only other domains to employ pinning without HSTS for 8 domains. In Twitter's case, these are mostly specific subdomains of twitter.com such as api.twitter.com and mobile.twitter.com, which suggests that support for legacy (non-HTTPS clients) is a significant motivation.

D. Dynamic HSTS

All sites with a preloaded HSTS policy should also be setting HSTS headers, otherwise security is limited to users using the Chrome browser as Firefox requires a header for inclusion in its preloaded list and other browsers (such as Safari, Opera) don't support preloaded HSTS at all. Still, 127 of the 390 preloaded HSTS domains (35.4%) *do not* set HSTS headers. Google domains account for about a fourth of these domains with 37 (64.9% of Google HSTS entries) not setting an HSTS header. It is unclear what Google's reasoning is for not setting HSTS headers for these domains, particularly as Google *does* set an HSTS header on 20 domains including accounts.google.com and plus.google.com. This represents a major security shortcoming as all of these domains are left vulnerable on non-Chrome browsers.

Additional dynamic HSTS domains: We also crawled all of the Alexa top 10,000 websites looking for additional domains setting HSTS headers. We observed only 156 sites setting an HSTS header (32 overlap with the preloaded HSTS list) and, of these dynamic HSTS sites, 17 are "opting out" by setting a `max-age=0`.

Even though only a small percentage are setting an HSTS header, a large percentage of the top 10,000 sites *should* be using HSTS. We found 866 sites using HTTP 30x codes to directly redirect from non-secure HTTP to HTTPS without setting an HSTS header, including many financial sites like citibank.com and chase.com, email providers like gmail.com and yahoo.com, and government sites like healthcare.gov. Redirecting from HTTP to HTTPS is a strong indication that

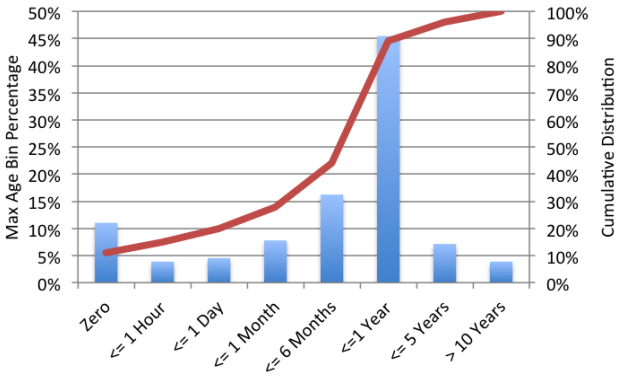

Fig. 2. Histogram and cumulative distribution of `max-age` values for dynamic HSTS headers.

these sites are intending to only offer their services over a secure connection; the fact that this remains so much more common than setting HSTS indicates developers may not be sufficiently aware of the new technology yet or may be nervous about causing some legacy clients to lose access. It's possible the redirects would not have happened if we had crawled with certain outdated `user-agent` strings and therefore this is an explicit policy choice. However, by redirecting without setting a header these sites are setting users up for HTTPS stripping every time they connect and HSTS is a much more secure choice.

Max-age values: Of the 156 valid HSTS headers we observed, there were a wide range of values for the `max-age` found in practice, as shown in Figure 2. The most common value is 31,536,000 seconds (one year), set by 44 domains. This is not surprising since this value is what is most often used as the example in the HSTS specification [21]. Outside of this standard, the values range widely with 13 headers set under 3600 seconds (a day) and 21 values over a year. Somo.vn sets the smallest value at just 60 seconds and www.npmjs.org sets the largest with a value of over 80 years. The `max-age` values also vary widely amongst the major players with Paypal only setting a header with an age of 4 hours and Twitter, Vine and Etsy all setting `max-age` over 20 years.

Very short values, such as the 8.2% of domains setting a value of less than 1 day, arguably undermine the value of HSTS. While they provide some protection against a passive eavesdropper in the case of developer mistakes (such as including an HTTP link), they are dangerous against an active attacker as many users will be repeatedly making untrusted initial connections if they do not visit the site very frequently.

It's an open question if very large `max-age` values will cause problems in the future if a small number of clients have very old cached policies. Unlike the proposal for HPKP, there is no standard maximum for `max-age` and user agents really are meant to cache the policy for 80 years if instructed to do so, although most browser instances in practice won't be around for that long.

Finally, we saw 17 sites setting a `max-age` of 0 including several big domains like Yahoo, LinkedIn, and t.co, a Twitter content delivery domain. Yahoo actually redirects from HTTP to HTTPS on every tested Yahoo subdomain but still specifically avoids using HSTS and intentionally sets a max age of 0. This is valid according to the specification [21] and is intended to intentionally tell clients to forget their cached policy for the domain, in case the domain wishes to revert to using HTTP traffic. Of these, 8 redirect from HTTPS to HTTP, clearly using the specification as intended.

TABLE IV. DYNAMIC HSTS ERRORS

	Alexa top 10k		Preloaded domains	
	Domains	%	Domains	%
Attempts to set dynamic HSTS	159	—	252	—
Doesn't redirect HTTP→HTTPS	32	20.1%	7	2.8%
Sets HSTS header only via HTTP	3	1.9%	0	0%
Redirects to HTTP domain	15	15%	15	6.0%
HSTS Redirects to non-HSTS	4	2.5%	15	6.0%
Malformed HSTS header	10	6.3%	3	1.2%
`max-age` = 0	17	10.7%	0	0.0%
0 < `max-age` < 1 day	13	8.2%	2	0.8%
Sets HSTS securely w/o errors	115	72.3%	229	90.9%

E. HSTS errors

Of the sites setting dynamic HSTS, a number did so in erroneous ways A summary of our findings is in Table IV. It is striking that overall, of the non-preloaded sites attempting to set HSTS dynamically, nearly 30% had a major security problem which undermined the effectiveness of HSTS. The rate of errors was significantly lower among sites with a preloaded HSTS policy.

HSTS sites failing to redirect: The HSTS specification [21] states that HSTS domains *should* redirect HTTP traffic to HTTPS. However, of the 139 sites with non-zero `max-ages`, 32 do not redirect. In addition, 30 preloaded HSTS sites do not redirect from HTTP to HTTPS. This also represents a security hole as first-time visitors may never transition to HTTPS and therefore never learn the HSTS policy.

HSTS headers set over HTTP: Another sign of confusion about the technology is sites setting an HSTS header over HTTP. Despite the HSTS standard specifies that this has no effect and should not be done [21], we observed 39 domains setting an HSTS header on the HTTP version of their sites. For sites otherwise implementing HSTS properly, this is a harmless mistake; however, we found 3 domains set HTTP HSTS headers without specifying a HTTPS header, a strong indication of the site misunderstanding the specification. 10 of the total HTTP HSTS domains (including popular tech sites like blockchain.info and getfirebug.com) also failed to redirect to HTTPS, indicating they may not understand HSTS does not achieve redirection on its own. In addition, 32 preloaded HSTS domains set HSTS headers over HTTP including the pinned site CryptoCat, but these sites all also set valid HTTPS HSTS headers. These sites are clearly attempting to improve the security of there connection via HSTS but perhaps misunderstanding the nature of the technology. The relatively high proportion of this error compared to sites successfully deploying HSTS (24.5%) is a clear sign of significant developer confusion.

Malformed HSTS: We also found 10 sites setting malformed HSTS headers including notable sites like paypal.com[13] and www.gov.uk. The most common

[13]The error at PayPal was particularly interesting as a PayPal engineer was the lead author of the HSTS standard [21].

mistake was including more than one value for `max-age`. For example, www.gov.uk's header includes the key-value pair `strict-transport-security: max-age=31536000, max-age=31536000, max-age=31536000;`. In all cases of multiple `max-ages`, the `max-age` value was the same. We confirmed Chrome and Firefox will tolerate this mistake without harm, but this technically violates the spec. We also found 3 setting simply a value without the required key `max-age=`. For example, www.pringler.com sets the header `strict-transport-security: 2678400`). This results in the header being ignored by the browser.

HSTS redirection to HTTP: We also observed 8 sites in the Alexa crawl that correctly set an HSTS header via HTTPS but then attempting to immediately redirect to HTTP (where several then set a meaningless HSTS header). For example, https://www.blockchain.info sets an HSTS header while redirecting to http://blockchain.info. HSTS-compliant browsers handle this error and instead to go to https:/blockchain.info which responds successfully, but a non-HSTS-complaint browser would be redirected back to HTTP which is clearly not the sites intention. In addition, 15 preloaded HSTS domains use 30x redirects from the listed preloaded domain to HTTP, 5 of which redirect back to HTTP versions of the same base domain which is still protected by the preloaded list. The Minnesota state website, mnsure.org, is both preloaded HSTS and sets a dynamic HSTS header but still responds with a 301 code (moved permanently) back to http://www.mnsure.org/.

V. Mixed content

Browsers isolate content using the *same-origin policy*, where the origin is defined as the scheme, host, and port of the content's URL. For example, the contents of a page loaded with the origin `a.com` should not be accessible to JavaScript code loaded by the origin `b.com`. This is a core principle of browser security dating to the early development of Netscape 2.0 [33] and formally specified in the years since [34]. Because HTTP and HTTPS are distinct schemes, the same-origin policy means content delivered over HTTPS is isolated from any insecure HTTP content an attacker injects with the same host and port. Therefore, an attacker cannot simply inject a frame with an origin of `http://a.com` into the browser to attempt to read data from `https://a.com`.

However, subresources such as scripts or stylesheets inherit the origin of the encapsulating document. For example, if `a.com` loads a JavaScript library from `b.com`, the code has an origin of `a.com` regardless of the protocol used to load it and can read user data (such as cookies) or arbitrarily modify the page contents. When an HTTPS page loads resources from an HTTP origin, this is referred to as *mixed content*. Mixed content is considered dangerous as the attacker can modify the resource delivered over HTTP and undermine both the confidentiality and integrity of the HTTPS page, significantly undermining the benefits of deploying HTTPS. For this reason, Internet Explorer has long blocked most forms of mixed content by default, with Chrome in 2011 and Firefox in 2013 following suit [35], although the details vary and there is no standard. Other browsers (such as Safari) allow mixed content with minimal warnings.

Not all mixed content is equally dangerous. While terminology is not standardized, mixed content is broadly divided into *active* content such as scripts, stylesheets, iframes, and Flash objects, which can actively modify the contents of the encapsulating page's DOM [36], and *passive* or *display* content such as images, audio or video. All browsers allow passive mixed content by default (usually modifying the graphical HTTPS indicators as a result). The distinction between active and passive content is not standardized. For example, XMLHttpRequests (Ajax) and WebSockets are considered passive content by Chrome and not blocked, but are blocked by Firefox and IE.

A. Pinning and mixed content

Unfortunately, the mixed content problem repeats itself with pinned HTTPS (as it has for HTTPS with Extended Validation certificates [37] and other versions of HTTPS with increased security). If a website served over a pinned HTTPS connection includes active subresources served over traditional (non-pinned) HTTPS, then just as with traditional mixed content an attacker capable of manipulating the included resources can hijack the encapsulating page. In the case of non-pinned mixed content, manipulation requires a rogue certificate instead of simply modifying HTTP traffic. It should be noted that this risk is not exactly analogous to traditional mixed content because an attacker's ability to produce a rogue certificate may vary by the target domain whereas the ability to modify HTTP traffic is assumed to be consistent regardless of domain. Still, including non-pinned content substantially undermines the security benefits provided by pinning.

A further potential issue with pinned content is that subresources may be loaded over pinned HTTPS with a different pinned set. This also represents a potential vulnerability, as an attacker may be able to obtain a rogue certificate satisfying the subresource's pin set but not the encapsulating page. Thus, the *effective pin set* of the encapsulating page is the union of the pin sets of all (active) subresources loaded by the page. If any of the subresources are not pinned, security of the page is reduced to the "implicit" set of pins consisting of all trusted root CAs, negating the security benefits of pinning completely.

Empirical results: Overall, from the homepages of 268 total base domains with a pinning policy, we observed a total of 8974 non-pinned resources being included across 165 domains. Of these, 1130 resources at 5 domains (dropbox.com, twitter.com, doubleclick.net, crypto.cat, and torproject.org) were active content. As noted above, this effectively negates the security goals of pinning for these domains.

While only 5 of 268 pinned base domains having active mixed-pinning content appears low at first glance, recall that 260 of the pinned domains are operated by Google. Google has been diligent to avoid mixed content and has the advantage of using its own content-delivery and advertising networks. The fact that 4 of the other 8 suffered from fatal active-mixed content problems suggests this will be a serious problem as pinning is incrementally deployed, particularly as these sites are on the cutting-edge of security awareness.

Sources of mixed content: A summary of the types of resources involved in mixed content errors is provided in Table V. The errors generally arise from including active

TABLE V. SELECTED TYPES OF PINNED MIXED CONTENT RESOURCES

	content type	#
active	`font`	56
	`link` (rel="stylesheet")	126
	`script`	655
	`subdocument`	15
	`stylesheet`	278
	total	1130
passive	`link` (rel="apple-touch-icon")	92
	`link` (rel="shortcut icon")	23
	`image`	7727
	total	6597

web analytics or advertising resources. For example, crypto.cat loads 27 scripts on 4 pages (including crypto.cat) from get.clicky.com, a Web analytics site.

Content delivery networks were another major source of errors. At dev.twitter.com we observed 85 loads from various subdomains of the content-delivery network akamai.net. Dropbox was responsible for 921 of the mixed-content loads we observed, including loading scripts, stylesheets (CSS), and fonts (considered active content) from cloudfront.net, another content-delivery network. They load these resources multiple times on essentially every page we visited within the domain.

Tor's homepage (specifically its blog) provided an ironic example. They include video (in the form of an iframe, considered active content) from YouTube, which is normally pinned, but used the third-party proxy site www.youtube-nocookie.com which prevent users' cookies from being sent to YouTube. Unfortunately by using this proxy site they lost the benefits of YouTube's pinning.

We also observed interesting errors in "widget" iframes for pinned sites which we happened to observe embedded in other pages in our crawl. For example, Twitter's embeddable gadget twitter.com/settings/facebook/frame loads (3 times) scripts from connect.facebook.net. Similarly, we observed the advertising network DoubleClick loading an assortment of advertising scripts from various locations within an iframe embedded in other sites. While this is meant to be included as an iframe at other sites, the non-pinned scripts it loads could still in some cases steal cookies and read user data. In particular, all of DoubleClick's cookies and many of Twitter's are not market `httponly` and can therefore be read by malicious scripts.

Impact of subdomains: A large number of these mixed content errors were due to resources loaded from subdomains of pinned domains without `includeSubDomains` set. Of the 8 pinned non-Google base domains, 3 domains had mixed content issues from loading a resource from a non-pinned subdomain of an otherwise pinned domain. Overall, and 97.06% of the unpinned active content loads were "self-inflicted" in that they were loaded from the same base domain.

Twitter had perhaps the most issues including loading scripts from syndication.twitter.com. Although they did set a dynamic HSTS Header to protect this resource load from this non-preloaded subdomain, this doesn't fix the fact that the domain isn't pinned. Tor also included content from numerous non-pinned subdomains. Dropbox and CryptoCat both link to their blog and forum subdomain without an HSTS header, and dropbox.com loads images and other passive resources from photo-*.dropbox.com without HSTS being set The blog.x.com subdomain was the most frequent subdomain with this issue with two of the five domains introducing "self-imposed" mixed content on this subdomain. These findings suggest that confusion over the relationship between subdomains owned by the same entity is a major source of errors and that developers may be forgetting when `includeSubDomains` is in effect.

Some of the problems may also simply come from modern websites being complicated and it being difficult to remember what is pinned and what isn't. To this end we observed many cases of content included from non-pinned domains owned by the same domain in practice, though not strictly subdomains. For example, Google loads images from *.ggpht.com on many of the major Google domains including play.google.com.

Expanded pin set mixed content: We observed 2072 references to resources protected by a *different* pin set from 6 domains. As discussed above, this expands the effective pin set to the union of the top-level page and all resources loaded. Of these, 733 were loaded as active content by 2: Twitter and Dropbox. Twitter accounted for 0 of these resources referencing platform.twitter.com. Since Twitter has two separately listed pin sets, it frequently increases its effective pin set size by loading content from the twitterCDN pin set on a twitterCom pin set domain. Both domains also include a script from ssl.google-analytics.com in multiple places. While this is a lower risk than including unpinned content, these findings support our expectation that mixed content handling will be more complicated for pinned content due to the multiple levels of potential risk..

Plain HTTP resources loaded by pinned domains: We observed a further 31939 references to resources over plain HTTP from 199 pinned domains. Only one domain, (doubleclick.net), made the mistake (observed 3 times) of including active content over HTTP by including a script from http://bs.serving-sys.com/. Again, this script was only loaded in a doubleclick.net iframe we observed within another page.

These numbers serve as a useful baseline for comparison and suggest that errors due to mixed pinning, particularly active content, are more common than mixed HTTP content. This suggests that this problem is not yet widely understood or appreciated, although it can completely undermine pinning.

B. HSTS Mixed Content

We also briefly consider the existence of mixed content between HSTS-protected HTTPS pages and non-HSTS resources loaded over HTTPS. Unlike the case of pinning, this is not a significant security risk at present because resources referenced via a URL with the `https` scheme must be accessed over HTTPS, even if they are not protected by HSTS.

There is an edge case which is not clearly defined by the specification [21] related to error messages. The HSTS standard requires hard failure with no warning if a connection to an HSTS domain has a certificate error, but doesn't specify if warnings can be shown for non-HSTS resources loaded by the page. This is likely a moot point, as modern browsers now typically block active content which would produce a certificate error even from non-HSTS pages.

Still, we found references to non-HSTS resources from HSTS pages were widespread, with 76550 references from 102 base domains, of which 32962 from 93 domains were

active content. As with the pinned mixed content errors, the vast majority were "self-inflicted" in that they were resources loaded from a common base domain, accounting for 99.80% of all mixed content and 99.84% of the active mixed content. Resources from explicit subdomains were again a major source of mixed policy, with 15310 references from 16 base domains, of which 6275 were active content.

VI. Cookie theft

A long-standing problem with the web has been the inconsistency between the same-origin policy defined for most web content and that defined for cookies [38]–[40]. Per the original cookie specification [38], cookies are isolated only by host and not by port nor scheme. This means cookies set by a domain via HTTPS will be submitted back to the same domain over HTTP [41]. Because cookies often contain sensitive information, particularly session identifiers which serve as login credentials, this poses a major security problem. Even if a domain `secure.com` only serves content over HTTPS, an active attacker may inject script into any page in the browser triggering an HTTP request to `http://secure.com/non-existent` and the outbound request will contain all of the users cookie's for the domain.

A. `Secure` cookies

To address this problem, the `secure` attribute for cookies was added in 2000 by RFC 2965 [39], the first update to the cookie specification. This attribute specifies that cookies should only be sent over a "secure" connection. While this was left undefined in the formal specification, all implementations have interpreted this to limit the cookie to being sent over HTTPS [40]. A persistent issue with the `secure` attribute is that it protects read access but not write access. HTTP pages are able to overwrite (or "clobber") cookies even if they were originally marked `secure`.[14]

B. Interaction of `secure` cookies and HSTS

At first glance, it may appear that HSTS supersedes and obviates the `secure` cookie attribute, as if a browser learns of an HSTS policy and will refuse to connect to a domain over plain HTTP at all it won't be unable to leak a secure cookie over HTTPS. Unfortunately, there is a confusing clash of options relating to subdomains. Cookies may include a `domain` attribute which specifies which domains the cookie should be transmitted to. By default, this includes all subdomains of the specified domain, unlike HSTS which does not apply to subdomains by default. Even more confusingly, the only way to limit a cookie to a single specific domain is to not specify a `domain` parameter at all, in which case the cookie should be limited to exactly the domain of the page that set it. However, Internet Explorer violates the standard [39] in this case and scopes the cookie to all subdomains anyway [41].

Thus, a well-intentioned website may expose cookies by setting HSTS but not the `secure` attribute if the HSTS policy does not specify `includeSubDomains` (which is the default) and the cookie is scoped to be accessible to subdomains (which occurs whenever any `domain` attribute is set). For example, suppose `a.com`, a domain which successfully sets HSTS without `includeSubDomains`, sets a cookie `session_id=`x with `domain=a.com` but does not set `secure`. This cookie will now be transmitted over HTTP to any subdomain of `a.com`. The browser won't connect over HTTP to `a.com` due to the HSTS policy, but will connect to `http://nonexistent.a.com` and leak the cookie value x over plain HTTP.

[14] In fact, HTTP connections can set cookies and mark them as `secure`, in which case they won't be able to read them back over HTTP.

TABLE VI. Pinned cookies vulnerable to theft

domain	domain hole	total cookies	insecure cookies
crypto.cat	*.crypto.cat	3	3
dropbox.com	*.dropbox.com	21	13
gmail.com	*.gmail.com	3	3
google.com	*.play.google.com	41	19
torproject.org	*.torproject.org	2	1
twitter.com	*.twitter.com	33	25
	total	103	64

An active attacker can inject a reference to `http://nonexistent.a.com` into any page in the browser, therefore this cookie is effectively accessible to any network attacker despite the domain's efforts to enforce security via HSTS. Thus we consider this to be a bug as it very likely undermines the security policy the domain administrator is hoping to enforce. HSTS does not serve as an effective replacement for `secure` cookies for this reason and it is advisable that HSTS sites generally mark all cookies as `secure` unless they are specifically need by an HTTP subdomain.

Empirical results

We checked for cookies vulnerable to theft over HTTP in our crawl of all preloaded HSTS domains in the Chrome preload list. We observed a total of 428 Cookies from 61 Base Domains which were accessible to non-HSTS subdomains. The vast majority of these, 352 (82.2%), did not set the `secure` attribute meaning they are vulnerable to theft by an active network attacker.

This issue was present on several important domains like Paypal, Lastpass and Usaa, and the cookies included numerous tracking and analytics cookies, user attributes cookies like county code and language, and unique session identification cookies like "guest_id," "VISITORID", and "EndUserId". Stealing these cookies can be a violation of user's privacy and may be used to obtain a unique identifier for users browsing over HTTPS. Encouragingly, however, all authentication cookies we were able to identify for these sites were marked as `secure` and hence could not be leaked over HTTP. This suggests that the `secure` attribute is relatively well-understood by web developers.

C. Interaction of cookies and pinning

A similar issue exists for pinned domains whereby a cookie may leak to unprotected subdomains. For example, if `a.com`, a pinned domain without `includeSubDomains`, sets a cookie `session_id=`x with `domain=a.com` the cookie will be transmitted over unpinned HTTPS to any subdomain of `a.com`. Note that even setting the `secure` flag doesn't help here—this will only require the cookie to be sent over HTTPS, but an attacker able to compromise HTTPS with a

rogue certificate will still be able to observe the value of the cookie.

Because there is no equivalent attribute to `secure` requiring a cookie to be sent over a pinned connection, currently there is no good fix for this problem. The only way to securely set cookies for a pinned domain is either to limit them to a specific domain (by not setting a `domain` parameter) or to specify `includeSubDomains` for the pinning policy.

Empirical results

We checked for cookies vulnerable to theft over non-pinned HTTPS from all pinned domains in the Chrome preload list. We observed 103 cookies on 6 pinned domains which are accessible by non-pinned subdomains, as summarized in Table VI. As mentioned above, there is no equivalent of the `secure` attribute to limit cookies to transmission over a pinned connection, and therefore all of these cookies are vulnerable.

Interestingly, the majority of these cookies are also in fact vulnerable to theft over plain HTTP as 64 of these cookies (62.1%) did not set the `secure` attribute.[15] This suggests that even if an attribute existed to limit cookies to a pinned connection, the semantics of this problem are complex enough that developers may not always deploy it.

Unlike our results for HSTS domains, we did observe authentication cookies vulnerable to theft. In the case of Twitter, the critical `auth_token` cookie could be leaked without pinned, which is sufficient to login successfully to a victim's Twitter account. Google and Dropbox did not have single session cookies[16] but we confirmed that the set of cookies leaked by both Dropbox and Google were sufficient for authentication. Thus we found authentication credentials were vulnerable at all three pinned domains with a login system.

Google's case was complex and interesting. While the majority of Google's pinning entries set `includeSubDomains` including google.com and thus would appear to avoid this error, play.google.com does not set `includeSubDomains`.[17] For subdomains *.play.google.com, the play.google.com entry overrides the less specific google.com entry as per RFC 6797 [21]. As a result, any subdomain of play.google.com like evil.play.google.com would not be bound by the Google pin set and an adversary with a rogue certificate for one of these domains would have access to all of cookies scoped for *.google.com there.

Recommendation to browsers

As a result of our findings with pinned cookies, we recommend that browser vendors extend the semantics of the `secure` attribute for cookies as follows: if a cookie is set by a domain with a pinning policy and marked `secure`, the cookie should only be transmitted over HTTPS connections which satisfy the pinning policy of the domain setting the cookie. This is a relatively simple fix which would close the security holes we found without introducing any new syntax. This can also reasonably be viewed as a valid interpretation of the original specification for `secure`, which never limited the syntax to mean simply HTTPS. Given that a large number of developers have successfully learned to mark important cookies as `secure`, it makes to extend this in a natural way as pinning and other HTTPS upgrades are deployed.

[15]Note that because the Chrome preload file requires only one line per domain for both pinning and HSTS policies, every domain with a non-`includeSubDomains` pinning policy also has a non-`includeSubDomains` HSTS policy (if any HSTS policy at all). A patch is currently pending to allow the two policies to be specified at different granularities.

[16]Identifying exactly which set of cookies is sufficient to hijack a user's session can be a difficult problem [42].

[17]In the transport_security_state_static.json (the preloaded HSTS site), the line includes the comment "play.google.com doesn't have include_subdomains because of crbug.com/327834;" however, this link is no longer valid indicating the original bug may have been fixed.

VII. Insecure linking

HSTS domains are vulnerable to HTTPS stripping prior to the user making a successful initial connection if they don't have a policy preloaded in the browser. Of course, many HSTS-compliant user agents don't have any preloaded policies. Thus, it is critical that all references to HSTS domains should be made with URLs declaring `https` as the protocol, even if the domain has a preloaded policy in some browsers.

This is generally best practice for any site implementing HTTPS. However, because setting an HSTS header is a strong signal that a domain should be reached via HTTPS only, we can consider any reference to an HSTS domain via an HTTP URL to be an error. With URLs for page resources, this error results in mixed content as discussed in Section V and can be blocked by browsers. However, hyperlinks (`a` tags) with an HTTP URL cannot generally be blocked by the browser like mixed content because there will always be legitimate cases where HTTP-only sites must be linked to over HTTP.

Getting the protocol correct on hyperlinks is essential for the end-to-end security model of HSTS with a mix of preloaded and header-declared policies. If users initially browse mainly at large sites whose HSTS status can be preloaded (such as search engines, webmail providers and social media sites) and mostly navigate to non-preloaded HSTS domains by means of HTTPS links from these preloaded domains, they will be protected from HTTPS stripping throughout. HTTP URLs, however, break the chain and offer an opportunity for a network attacker to block the HTTP to HTTPS redirect. In this section we examine the prevalence of dangerous "weak links" to HSTS domains.

A. Insecure links to HSTS domains

In our depth-1 crawling of all home-pages found in the preloaded list, we observed 6666 non-secure links from 199 domains to preloaded HSTS sites. For comparison, 378012 links to preloaded HSTS sites were correctly served with the `https` protocol. Thus, 1.7% of links are served incorrectly, each of which is a security hole.

Breaking the numbers down further though, of 363130 links between HSTS pages with a common base domain, 4461 were over HTTP. For example, https://play.google.com/store links to http://play.google.com and https://wiki.python.org links to http://www.python.org

Of the remaining 14882 links to HSTS sites which were cross-domain, 2204 were HTTP links. These links are the most

TABLE VII. TOP 10 HSTS DOMAINS LINKED TO OVER HTTP

	domain	# links	subdomain	# links
1	google.com	1250	code.google.com	1119
			groups.google.com	93
			plus.google.com	34
			mail.google.com	2
			sites.google.com	1
			play.google.com	1
2	twitter.com	602	twitter.com	583
			www.twitter.com	19
3	bitbucket.org	45	bitbucket.org	45
4	onedrive.com	41	blog.onedrive.com	29
			dev.onedrive.com	12
5	paypal.com	26	www.paypal.com	26
6	publications.qld.gov.au	20	publications.qld.gov.au	20
7	python.org	17	www.python.org	13
			python.org	2
			docs.python.org	2
8	eff.org	6	www.eff.org	6
9	dropbox.com	5	dropbox.com	3
			www.dropbox.com	2
10	gmail.com	3	www.gmail.com	2
			gmail.com	1
		total		2024

important, as they might lead a user to discover a new HSTS domain provide an opportunity for an attacker to intercept a potential upgrade to HTTPS. Because a substantial portion of them (12.9%) were HTTP, this suggests insecuring linking is a widespread problem on the web.

A summary of the link targets for cross-domain links is shown in Table VII and mostly consisted of external websites linking to a few popular domains. We also observed persistent errors from national versions of the Google homepage (like www.google.dj) linking to maps., www. and translate.google.com subdomains. Most Google pages also reference support.google.com in a non-secure manner.

B. Insecure links to pinned domains

Unlike the case for HSTS domains, there is currently no way to securely specify a link to pinned domains. All links to pinned domains are inherently "weak" in that they rely on the browser having a preloaded pinning policies for protection. Of course, there are no dynamically pinned domains yet, but they are expected to arrive soon which will make this a practical security concern. Several proposals have been put forward for extending URL or HTML syntax to enable specifying key pins or other security policies in hyperlinks, such as YURLs [43] or S-links [44], although there are complex concerns about the interaction of these schemes with the same-origin policy [20]. The risks outlined in this section suggest that securely deploying dynamic pinning will require a secure linking scheme may need to be added to protect initial connections to pinned domains.

VIII. RELATED WORK

A. Empirical web security studies

Our work fits nicely into a long and fruitful line of measurement studies of web security, covering a wide variety web security topics such as authentication cookies [45], `http-only` cookies [42], password implementations [46], third-party script inclusions [47], [48], third-party trackers [49], [50], Flash cross-domain policies [51] and OpenID implementations [52], [53]. A classic problem is detecting cross-site scripting vulnerabilities which is practically its own sub-field of research [54]–[57]. A common model for this research is exploration and analysis of an emerging threat on the web, followed by measurement and crawling to detect its prevalence and character. A desirable research outcome is for automated detection to be built-in to web application security scanners [58]–[60]. Ultimately, browsers themselves have come to absorb some of this logic, for example, Chrome and Firefox both now contain warnings to developers about basic mixed-content errors in their built-in developer consoles.

Several issues identified in our research are solid candidates for inclusion in future automated tools; in particular cookies vulnerable to theft, pinning mixed content, some types of erroneous or short-lived HSTS headers, and insecure links to HSTS domains. As a first step we plan to release our own analysis code as an open-source add-on for WPM and Selenium.

B. Empirical studies of HTTPS and TLS

A significant amount of research has also focused specifically on empirical errors with HTTPS and TLS, of which Clark and van Oorschot provide the definitive survey [3]. Important studies of cryptographic vulnerabilities have included studies of key revocation after the Debian randomness bug [61], studies of factorable RSA keys due to shared prime factors [62], [63], studies of elliptic curve deployment errors in TLS [64], forged TLS certificates in the wild [17] and multiple studies of key sizes and cipher suites used in practice [23], [65], [66]. Our work is largely distinct from these in that we focus on two new aspects of HTTPS (pinning and strict transport security) which are vulnerabilities at the HTTPS (application) level rather than the TLS (cryptographic) level.

Perhaps the most similar study to ours was by Chen et al. [67] which focused on mixed-content and stripping vulnerabilities. This study was performed prior to the advent of HSTS, pinning, and mixed content blocking. Hence this work can be viewed as a first-generation study compared to our second-generation study based on newly introduced security measures (although no doubt many of the original vulnerabilities are still present on the web today).

Other empirical studies of the TLS ecosystem have focused on certificate validation libraries [68], non-browser TLS libraries [69], the interaction of HTTPS with content-delivery networks [70] and TLS implementations in Android apps [71], [72]. Again, these efforts all found widespread weaknesses. A common theme is developers not correctly understanding the underlying technology and using them in an insecure manner.

C. Other proposals for improving HTTPS

We briefly overview several noteworthy proposals for further improving HTTPS. These proposals are mainly aimed at improving the certificate model on the web and limiting the risk of rogue certificates. Hence they could compliment or supplant key pinning. HSTS is generally considered an acceptable solution to HTTPS stripping. We exclude other proposals for improving web security such as Content Security Policy (CSP) [73].

1) DANE: DNS-based Authentication of Named Entities (DANE) [74] is a proposal for including the equivalent of public key pins in DNS records, relying on DNSSEC [75] for security. This has the advantage of avoiding the scalability concerns of preloaded pins and potentially being easier[18] and more efficient[19] to configure than pins set in headers. Unfortunately, DANE adoption is held up pending widespread support for DNSSEC, which no common browsers currently implement.

DANE does not currently contain support for declaring policies applicable to all subdomains. Based on our study, we would strongly advise such support be added (and possibly turned on by default) to avoid the type of mixed content and cookie vulnerabilities we observed with pinning today. On the positive side, DANE solves the problem of weak links as user agents must resolve a new domain and therefore fetch its policy prior to the initial connection.

2) Out-of-chain key pinning: An alternative to pinning to keys within a cite's certificate chain is to specify a separate self-managed public key which must sign all end-entity public keys, in addition to requiring a certificate chain leading to a trusted CA. This avoids fully trusting any external CA while offering more flexibility than pinning to an enumerated set of end-entity public keys. Conceptually, it is similar to pinning to a domain-owned key in a CA-signed, domain-bound intermediate certificate.[20] TACK [76] proposes distributing out-of-chain public keys using continuity,[21] while Sovereign Keys [77] proposes using a public append-only log.

TACK explicitly does not contain support for extending policies to subdomains, instead recommending that implementers omit the domain parameter to limit cookies to one domain and/or and set the `secure` flag to avoid cookie theft. Of course, the `secure` flag is inadequate for defending against rogue certificates given that one cannot set a TACK policy for non-existent subdomains. Sovereign Keys, by contrast, does support wildcards to extend support to subdomains.

3) Public logging: Due to the risk of improperly configured pinning policies causing websites to be inaccessible, some proposals aim simply to require that all valid certificates are publicly logged to ensure rogue certificates are detected after the fact. Certificate Transparency [78] (CT) is the most prominent of these efforts, recording all valid end-entity certificates in a publicly verifiable, append-only log. As currently proposed, clients will begin to reject certificates lacking proof of inclusion in the log after universal adoption[22] by all public CAs.[23] A somewhat-related proposal is Accountable Key Infrastructure [79].

As proposed, Certificate Transparency would avoid most of the subtle issues identified in this work in that the burden on web developers is extremely low (the only requirement is to start using a CT-logged certificate prior to some future deadline). Given the number of errors we observed in our study, this seems like a major design advantage. However, if CT is not able to be adopted via the proposed "flag day" strategy, it may be necessary to distribute policies to browsers specifying which domains require CT protection. This problem would be largely similar to distributing HSTS today. Thus the lessons from our paper would apply to designing such a mechanism, which most likely would be implemented as an extra directive in HSTS itself.

IX. Concluding discussion

HSTS is in the early stages of adoption and pinning is in the very-early stages of adoption. Both technologies though have already greatly enhanced security at a number of websites. It is a tribute to pinning that it has been the first mechanism to detect most CA compromises since it was deployed in 2010. Still our research shows that in early deployment a large number of misconfiguration errors are undermining the potential security in many cases.

Some of this can be attributed to developer unfamiliarity with the new tools. Most prior work has found that some substantial portion of web developers will shoot themselves in the foot given any new technique to do so. It is worth emphasizing however that many of the errors we observed were made by large websites that are in many ways at the forefront of web security. Our sample was biased towards the very developers who should be in the best position to understand these new tools. While developer familiarity may increase with time, if the new techniques gain more widespread adoption this may also mean less security-aware developers begin to deploy them.

This should serve as a reminder that simplicity for developers is a critical aspect of any web security technology. Considering the root causes of some of the errors we found, better defaults might have helped. For example, we might advocate a default value of perhaps 30 days for HSTS policies set without an explicit max-age. Forcing all developers to choose this has probably lead to unwise choices on both ends of the spectrum in addition to malformed headers. Setting sensible defaults for pinning is far more challenging—there is no clear way to choose a default "backup pin" besides the values currently being used. This suggests that pinning may never be a simple "on switch" for developers, unless TLS certificate management can be abstracted away completely.

Another important takeaway from our work is that many developers appear to not fully understand the same-origin policy and the relation of subdomains to one another. This is particularly true with cookies. For both HSTS and pinning, having policies apply to subdomains by default, with an option to disable this or turn it off for specific subdomains, would be a safer design. Extending cookies' `secure` attribute to require pinning (where applicable) as well as HTTPS, as discussed in Section VI-C, also appears to be a simple step towards making the technology match developer expectations more closely.

[18] It is an open question whether web developers are more comfortable configuring HTTP headers or DNS TXT records.

[19] Unlike for HSTS which operates with relatively small headers, there is significant concern about the efficiency of setting pin in headers, which may be hundreds of bytes long for complicated pin sets as have been seen in practice in our study.

[20] Domain-bound intermediates are possible using X.509's `nameConstraints` extension. However, this extension is not universally supported and non-supporting clients will reject such certificates.

[21] Continuity in TACK is distinct from HSTS/HPKP in that clients only retain a TACK policy for as long into the future as the policy has been consistently observed in the past, subject to a maximum of 30 days.

[22] Even after universal adoption, clients must wait until all extant legacy certificates have expired to require CT proofs.

[23] Private CAs, such as those used within an enterprise, are excluded.

Finally, we might advocate for a streamlining of HTTPS security features to make configuration as central as possible. We should accept the fact that HSTS and pinning will not be the last HTTPS enhancements ever adopted. HSTS and pinning have merged together somewhat in browser preloads, but not in headers where two distinct syntaxes are being standardized. It would be useful to combine dynamic HSTS and pinning declarations into a more flexible and extensible syntax that developers can declare once, preferably with very sensible defaults, rather than have to learn new syntax and subtleties as each new patch is applied.

ACKNOWLEDGMENT

The authors would like to thank...

REFERENCES

[1] E. Rescorla, "HTTP over TLS," RFC 2818, Internet Engineering Task Force, 2000.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Internet Engineering Task Force, 2008.

[3] J. Clark and P. C. van Oorschot, "SSL and HTTPS: Revisiting past challenges and evaluating certicate trust model enhancements," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[4] M. Marlinspike, "New Tricks For Defeating SSL In Practice," in *Black Hat DC*, 2009.

[5] J. Hodges, C. Jackson, and A. Barth, "RFC 6797: HTTP Strict Transport Security (HSTS)," 2012.

[6] C. Soghoian and S. Stamm, "Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL," *Financial Cryptography and Data Security*, 2012.

[7] C. Evans, C. Palmer, and R. Sleevi, "Internet-Draft: Public Key Pinning Extension for HTTP," 2012.

[8] "Alexa: The Web Information Company," http://www.alexa.com, 2014.

[9] T. Dierks and c. Allen, "The Transport Layer Security (TLS) Protocol Version 1.0," RFC 2246, Internet Engineering Task Force, 1999.

[10] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, Internet Engineering Task Force, 2006.

[11] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101, Internet Engineering Task Force, May 2011.

[12] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[13] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," RFC 2459, Internet Engineering Task Force, 1999.

[14] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness," in *USENIX Security Symposium*, 2009, pp. 399–416.

[15] D. Akhawe and A. P. Felt, "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness," *USENIX Security Symposium*, pp. 257–272, 2013.

[16] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe?: Understanding TLS errors on the web," in *Proceedings of the 22nd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2013, pp. 59–70.

[17] L.-S. Huang, A. Rice, E. Ellingsen, and C. Jackson, "Analyzing forged ssl certificates in the wild," *IEEE Symposium on Security and Privacy*, 2014.

[18] A. Ornaghi and M. Valleri, "Man in the middle attack demos," *Blackhat Security*, 2003.

[19] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The Emperor's New Security Indicators," in *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[20] C. Jackson and A. Barth, "Beware of Finer-Grained Origins," *Web 2.0 Security and Privacy*, 2008.

[21] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," RFC 5246 (Proposed Standard), Internet Engineering Task Force, 2012.

[22] D. Eastlake, "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066, Internet Engineering Task Force, 2011.

[23] I. Ristic, "Internet SSL Survey 2010," *Black Hat USA*, vol. 3, 2010.

[24] P. Eckersley and J. Burns, "The (decentralized) SSL observatory (Invited Talk)," in *20th USENIX Security Symposium*, 2011.

[25] J. Kasten, E. Wustrow, and J. A. Halderman, "Cage: Taming certificate authorities by inferring restricted scopes," in *Financial Cryptography and Data Security*. Springer, 2013, pp. 329–337.

[26] A. Niemann and J. Brendel, "A Survey on CA Compromises," 2013.

[27] "OpenWPM project," *https://github.com/citp/OpenWPM*, 2014.

[28] J. Huggins and P. e. a. Hammant, "Selenium browser automation framework," *http://code.google.com/p/selenium*, 2014.

[29] A. Langley, "Strict Transport Security," Imperial Violet (blog), January 2010.

[30] ——, "Public Key Pinning," Imperial Violet (blog), May 2011.

[31] D. Keeler, "Preloading HSTS," Mozilla Security blog, November 2012.

[32] Google Support, "Block adult content at your school with SafeSearch," retrieved 2014.

[33] J. Ruderman, "The same origin policy," http://www.mozilla.org/projects/security/components/same-origin.html, 2001.

[34] A. Barth, "The Web Origin Concept," RFC 6454, Internet Engineering Task Force, 2011.

[35] I. Ristic, "HTTPS Mixed Content: Still the Easiest Way to Break SSL," Qualys Security Labs Blog, 2014.

[36] T. Vyas, "Mixed Content Blocking Enabled in Firefox 23!" Mozilla Security Engineering—Tanvi's Blog, 2013.

[37] M. Zusman and A. Sotirov, "Sub-prime PKI: Attacking extended validation SSL," *Black Hat Security Briefings, Las Vegas, USA*, 2009.

[38] D. Kristol, "HTTP State Management Mechanism," RFC rfc2109, Internet Engineering Task Force, 1997.

[39] D. Kristol and L. Montulli, "HTTP State Management Mechanism," RFC 2965, Internet Engineering Task Force, 2000.

[40] A. Barth, "HTTP State Management Mechanism," RFC 6265, Internet Engineering Task Force, 2011.

[41] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[42] Y. Zhou and D. Evans, "Why Arent HTTP-only Cookies More Widely Deployed?" *Proceedings of 4th Workshop on Web 2.0 Security and Privacy*, vol. 2, 2010.

[43] T. Close, "YURLs," http://www.waterken.com/dev/YURL/, United States, 2004.

[44] J. Bonneau, "S-links: Why distributed security policy requires secure introduction," in *Web 2.0 Security & Privacy*, May 2013.

[45] K. Fu, E. Sit, K. Smith, and N. Feamster, "Dos and don'ts of client authentication on the web," in *USENIX Security*. Berkeley, CA, USA: USENIX Association, 2001.

[46] J. Bonneau and S. Preibusch, "The password thicket: technical and market failures in human authentication on the web," *WEIS '10: Proceedings of the Ninth Workshop on the Economics of Information Security*, June 2010.

[47] C. Yue and H. Wang, "Characterizing insecure JavaScript practices on the web," in *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009, pp. 961–970.

[48] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 736–747.

[49] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012.

[50] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third–party tracking on the web," *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[51] A. Venkataraman, "Analyzing the Flash crossdomain policies," Master's thesis, 2012.

[52] S.-T. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of oauth sso systems," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 378–390.

[53] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 365–379.

[54] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *Telecommunications Energy Conference, 2004. INTELEC 2004. 26th Annual International*. IEEE, 2004, pp. 71–80.

[55] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *IEEE Symposium on Security and Privacy*. IEEE, 2006, pp. 6–pp.

[56] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 171–180.

[57] A. E. Nunan, E. Souto, E. M. dos Santos, and E. Feitosa, "Automatic classification of cross-site scripting in web pages using document-based and url-based features," in *Computers and Communications (ISCC), 2012 IEEE Symposium on*. IEEE, 2012, pp. 000 702–000 707.

[58] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *Proceedings of the 15th International Conference on World Wide Web*. ACM, 2006, pp. 247–256.

[59] M. Curphey and R. Arawo, "Web application security assessment tools," *IEEE Symposium on Security & Privacy*, 2006.

[60] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 566–571.

[61] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: results from the 2008 debian openssl vulnerability," in *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference*. ACM, 2009, pp. 15–27.

[62] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, Whit is right," *IACR Cryptology ePrint Archive*, vol. 2012, p. 64, 2012.

[63] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," in *USENIX Security Symposium*, 2012, pp. 205–220.

[64] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow, "Elliptic curve cryptography in practice." *IACR Cryptology ePrint Archive*, vol. 2013, p. 734, 2013.

[65] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011, pp. 427–444.

[66] B. Amann, M. Vallentin, S. Hall, and R. Sommer, "Revisiting SSL: A large-scale study of the internets most trusted protocol," Technical report, ICSI, Tech. Rep., 2012.

[67] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang, "Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments," in *IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 347–359.

[68] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations," *IEEE Symposium on Security and Privacy*, 2014.

[69] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 38–49.

[70] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu, "When https meets cdn: A case of authentication in delegated service," *IEEE Symposium on Security and Privacy*, 2014.

[71] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 50–61.

[72] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 49–60.

[73] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 921–930.

[74] P. Hoffman and J. Schlyter, "RFC 6698: The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA," IETF RFC 6698, 2012.

[75] G. Ateniese and S. Mangard, "A new approach to dns security (dnssec)," in *Proceedings of the 8th ACM Conference on Computer and Communications Security*. ACM, 2001, pp. 86–95.

[76] M. Marlinspike and T. Perrin, "Internet-Draft: Trust Assertions for Certificate Keys," 2012.

[77] P. Eckersley, "Internet-Draft: Sovereign Key Cryptography for Internet Domains," 2012.

[78] B. Laurie, A. Langley, and E. Käsper, "Internet-Draft: Certificate Transparency," 2013.

[79] T. H.-J. Kim, L.-S. Huang, A. Perring, C. Jackson, and V. Gligor, "Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure," in *Proceedings of the 22nd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2013.